Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage

Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Sergio Maffeis

Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage

Chetan Bansal[1], Karthikeyan Bhargavan[2⋆], Antoine Delignat-Lavaud[2], and Sergio Maffeis[3⋆⋆]

[1] BITS Pilani-Goa
[2] INRIA Paris-Rocquencourt
[3] Imperial College London

Abstract. To protect sensitive user data against server-side attacks, a number of security-conscious web applications have turned to client-side encryption, where only encrypted user data is ever stored in the cloud. We formally investigate the security of a number of such applications, including password managers, cloud storage providers, an e-voting website and a conference management system. We find that their security relies on both their use of cryptography and the way it combines with common web security mechanisms as implemented in the browser. We model these applications using the WebSpi web security library for ProVerif, we discuss novel attacks found by automated formal analysis, and we propose robust countermeasures.

Keywords: Web Security, Formal Methods, Protocol Verification

1 Application-level Cryptography on the Web

Many web users routinely store sensitive data online, such as bank accounts, health records and private correspondence. Servers that store such data are a tempting target for cybercrime: a single attack can yield valuable data, such as credit card numbers, for millions of users. As websites move to using cloud-based data storage, the confidentiality of user data and the trustworthiness of the hosting servers has come further into question.

Transport layer security (TLS) as provided by HTTPS [21] does not fully address these concerns. TLS protects sensitive data over the wire as it travels between a browser and a website. However, it does not protect data at rest, when it is stored on the client or the server, where it can be accessed by an attacker stealing a laptop or hacking into a server. To protect from these risks, web applications use a combination of application-level cryptography and browser-based security mechanisms to securely handle user data. Our goal is to formally investigate the effectiveness of these mechanisms and their concrete deployments.

⋆ Bhargavan is supported by ERC grant CRYSP
⋆⋆ Maffeis is supported by EPSRC grant EP/I004246/1.

Application-level cryptography. To protect data from hackers, websites like Dropbox [2] systematically encrypt all files before storing them on the cloud. However, since the decryption keys must be accessible to the website, this architecture still leaves user data vulnerable to dishonest administrators and website vulnerabilities. A more secure alternative, used by storage services like SpiderOak and password managers like 1Password, is *client-side encryption*: encrypt all data on the client before uploading it to the website. Using sophisticated cryptographic mechanisms, the server can still perform limited computations on the encrypted data [19]. For example, web applications such as ConfiChair [7] and Helios [4] combine client-side encryption with server-side zero-knowledge constructions to achieve stronger user privacy goals.

These application-level cryptographic mechanisms deserve close formal analysis, lest they provide a false sense of security to their users. In particular, it is necessary to examine not just the cryptographic details (i.e. what is encrypted), but also how the decryption keys are managed on the the browser.

Browser-based security mechanisms. Even with client-side encryption, the server is still responsible for access control to the data it stores. Web authentication and authorization typically rely on password-based login forms. Some websites use single sign-on protocols like OAuth [17] to delegate user authentication to third parties. After login, the user's session is managed using cookies known only to the browser and server. JavaScript is then used to interact with the user, make AJAX requests to download data over HTTPS, store secrets in HTML5 local storage, and present decrypted data to the user.

The security of the application thus depends on both the server and on browser-based mechanisms like cookies and JavaScript. That is dangerous, considering the prevalence of web vulnerabilities such as Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), open redirectors or phishing, even on major websites. In previous work, our survey of encrypted storage services [11] uncovered many such vulnerabilities and showed how can be exploited to bypass both client-side and server-side cryptographic protections. However, these attacks were found by manual inspection aided by tracing tools. Can we search for such attacks systematically and exhaustively? More importantly, can we evaluate any proposed countermeasures to ensure that they are not vulnerable to variations of the same attacks? In response to both these questions, we follow [8] in advocating the automated formal analysis of web security mechanisms.

Formal analysis of cryptographic web applications. Standard cryptographic attacker models employ a crude notion of compromise: if a client or server performs any action outside the description of the protocol, it is considered compromised. This characterization is too strong for web applications which may contain dozens of pages, among which only a few are security-sensitive. We need a new attacker model that allows honest websites to have *some* vulnerable pages.

In previous work [8], we proposed WebSpi, a formal model of web attackers and browser-based security mechanisms, written as a library for ProVerif [12]. We used WebSpi to analyze web authorization and single sign-on applications against a limited set of web attacks including CSRF and open redirectors. Here,

Name	Key Derivation	Encryption	Integrity	Metadata Integrity	Sharing
Wuala	PBKDF2	AES, RSA	HMAC	✓	✓(PKI)
SpiderOak	PBKDF2	AES, RSA	HMAC	✓	✓
BoxCryptor	PBKDF2	AES	None	✗	✗
CloudFogger	PBKDF2	AES, RSA	None	✗	✓(PKI)
1Password	PBKDF2-SHA1	AES	None	✗	✓
LastPass	PBKDF2-SHA256	AES, RSA	None	✗	✓
PassPack	SHA256	AES	None	✓	✓
RoboForm	PBKDF2	AES, DES	None	✗	✓
Clipperz	SHA256	AES	SHA256	✓	✗
ConfiChair	PBKDF2	RSA, AES	SHA1	✓	✓(PKI)
Helios	N/A	El Gamal	SHA256	Zero-Knowledge Proof	N/A

Table 1. Example encrypted web storage applications

we extend WebSpi to cover additional browser mechanisms such as local storage, AJAX, and the associated same origin policy, as well as to account for new attacks such as XSS, insecure cookies or JSONP-based CSRF.

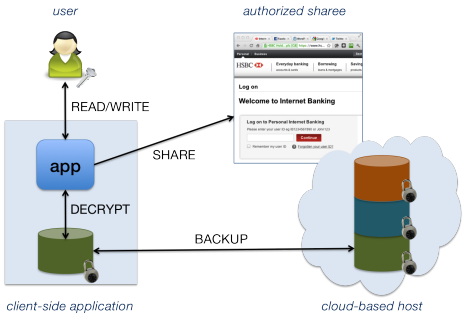
The analysis of [8] did not address cryptographic issues. Here we extend WebSpi to study a series of commercial and academic cryptographic web applications. Our analysis reveals several new web-based attacks that expose flaws in their cryptographic designs, and formally reconstructs attacks previously reported in [11]. These attacks have been responsibly disclosed, and most were fixed in accordance with our suggestions. Our formal analysis suggests new countermeasures that are more robust in the face of web vulnerabilities. We verify these designs against attackers modeled in WebSpi. In summary, our work extends the state of the art by combining symbolic cryptographic protocol analysis with a realistic web attacker model. All the WebSpi scripts referenced in this paper are available online at `http://prosecco.inria.fr/webspi/`.

Related Work A number of cryptographic protocols underlying real-world web applications have been verified for sophisticated security properties. Closely related to this paper are the symbolic analyses of ConfiChair [7], Helios [4], and Plutus [13]. However, none of these consider web attacks like CSRF and XSS, and as we show for ConfiChair and Helios, their security guarantees can be broken by such standard web vulnerabilities.

Various attacks have previously been found on encrypted storage applications: on their cryptographic design [10], on their web deployment [5], and on combinations of the two [11]. Such attacks are typically found using ad hoc tracing tools, and these works do not offer any positive guarantees for countermeasures. These attacks serve as motivation for our fomal analysis.

Several works propose formal models of browser-based security mechanisms [24, 15, 6, 16]. Closely related to our work are the models of [6], which capture many of the same web vulnerabilities, and can be analyzed using Alloy [18]. However, they do not generally consider cryptography, whereas our use of ProVerif enables a combination of cryptographic and web security analysis.

2 Encrypted Web Storage Applications



We study encrypted web storage, a core functionality of many security-conscious web applications. More specifically, we evaluate the design, implementation, and use of client-side encryption in the web applications of Table 1. The general architecture of such applications is depicted on the right. They fall in three categories:

File storage services, such as Wuala and SpiderOak, offer a remote encrypted backup folder synchronized across various user devices with options to share folders and files with non-registered users by sending web links.

Password managers, such as 1Password and LastPass, integrate with a browser to store user login credentials for different websites. When the user browses to a known website, the password manager offers to automatically fill in the login form. The password database is kept encrypted on the client and backed up remotely, and can be synchronized across the user's devices.

Privacy-conscious websites, such as ConfiChair for conference management and Helios for electronic voting use client-side encryption to protect users against powerful attackers who may obtain control over the website itself.

All these applications implement an encrypted storage protocol and then use it to build more advanced features. We begin by describing one such protocol.

2.1 An Encrypted Storage Protocol

Suppose a user u has some sensitive data db with metadata m that she wishes to backup on a storage server. For example, db may be a local file with name m, or db may contain a password for the website m. u uses some client software a to communicate with the server b. When u creates or modifies db, a encrypts the data and sends it to the storage server. Periodically, a downloads and synchronizes its local copy of the encrypted db with the storage server. u does not know or trust the storage server, we assume it is somewhere in the cloud. We describe these two protocols below.

Notation The cryptographic primitives crypt and $\mathsf{decrypt}$ represent symmetric encryption and decryption (e.g. AES in CBC mode); mac represents MACing (e.g. HMAC with SHA256); kdf represents password-based key derivation (e.g. PBKDF2). We model a TLS channel c with some server b as follows: an outgoing message m is denoted $\mathsf{TLS}_c^{\rightarrow b}(m)$ and an incoming message is denoted $\mathsf{TLS}_c^{\leftarrow b}(m)$.

Update and Synchronization protocols. Assume that u and b share a secret $\mathsf{secret}_{u,b}$ and that a has a local encryption key K and MAC key K' that it never sends to the server. These three secrets are stored on the client and may be encrypted under a password for additional security.

Update Cloud Storage: Update(u,m,db)

	a and b establish TLS connection c: $\mathsf{TLS}_c^{\rightarrow b}(-)$, $\mathsf{TLS}_c^{\leftarrow b}(-)$
1. $a \rightarrow b$	$\mathsf{TLS}_c^{\rightarrow b}(\mathsf{Authenticate}(u, \mathsf{secret}_{u,b}))$
	b verifies $\mathsf{secret}_{u,b}$ and associates c with u
	a updates encdb to (m,e=crypt K db,h=mac K' (m,e))
2. $a \rightarrow b$	$\mathsf{TLS}_c^{\rightarrow b}(\mathsf{Upload}(\mathsf{m}, \mathsf{e}, \mathsf{h}))$
	b updates storage[u] to (m,e,h)

In the protocol above, $\mathsf{Authenticate}(a, \mathsf{secret}_{a,b})$ denotes a tagged message requesting authentication of user u with password $\mathsf{secret}_{u,b}$. Similarly, message $\mathsf{Upload}(\mathsf{m}, \mathsf{e}, \mathsf{h})$ requests to upload the metadata m with the encryption e of the database db under the key K, and the MAC h of m and e under the MAC key K'. Hence, this protocol protects the confidentiality and ciphertext integrity of db, and the metadata integrity of m. Some applications in Table 1 do not provide metadata integrity; in Section 4.3 we show how this leads to a password recovery attack on 1Password.

The user data db is stored encrypted on the client. If an authorized user requests to read it, the client a will verify the MAC, decrypt encdb, and display the plaintext. The synchronization protocol authenticates the user, downloads the most recent copy of the encrypted database, and verifies its integrity.

Synchronize with Cloud Storage: Synchronize(u)

	a and b establish TLS connection c: $\mathsf{TLS}_c^{\rightarrow b}(-)$, $\mathsf{TLS}_c^{\leftarrow b}(-)$
1. $a \rightarrow b$	$\mathsf{TLS}_c^{\rightarrow b}(\mathsf{Authenticate}(u, \mathsf{secret}_{u,b}))$
	b verifies $\mathsf{secret}_{u,b}$ and associates c with u
	b retrieves storage[u] = (m,e,h)
3. $b \rightarrow a$	$\mathsf{TLS}_c^{\leftarrow b}(\mathsf{Download}(\mathsf{m}, \mathsf{e}, \mathsf{h}))$
	a checks that mac K' (m, e) = h
	a updates encdb to (m,e,h)

Attacker model. The protocols described above protect the user from *compromised servers*, *network attackers* and *stolen devices*. In particular: an attacker gaining control of a storage server, or of a device on which the client application is installed but not running, must be unable to recover any plaintext or information about user credentials; a user must be able to detect any tampering with the stored data; a network attacker must be unable to eavesdrop or tamper with communications through the cloud. Under reasonable assumptions on the cryptographic primitives, one can show that the reference protocol described above preserves the confidentiality of user data (see, for example [7]). However, such proofs do not reflect the actual deployment of web-based encrypted storage applications, leading to attacks that break the stated security goals, despite the formal verification of their cryptographic protocols.

2.2 Deploying Encrypted Storage Protocols over the Web

Although encrypted storage protocols can be deployed using custom clients and servers, a big advantage of deploying it through a website is portability. The storage service may then be accessed from any device that has a web browser without the need for platform-specific software. This raises the challenge that the developer now needs to consider additional web-based attack vectors that affect websites and browsers. Consider an encrypted storage protocol where the client a is a browser and the server b is a website. We discuss the main design questions raised by this deployment architecture.

Password-based Key Derivation. Browser a must be able to obtain the secret $\mathsf{secret}_{u,b}$ to authenticate to the server. Then it must be able to obtain the encryption key K and MAC key K'. The usual solution is that all three of these secrets are derived from a passphrase, sometimes called a master password. The key derivation algorithm (e.g. PBKDF2) typically requires a salt and an iteration count. Choosing a high iteration count *stretches* the entropy of the passphrase by making brute-force attacks more expensive, and choosing different salts for each user reduces the effectiveness of pre-computed tables [20]. In the following, we assume that each of the three secrets is derived with a different user-dependent constant (A_u, B_u, C_u) and a high iteration count (iter).

User Authentication and Cookie-based Sessions. To access a storage service a user must log in with the secret $\mathsf{secret}_{u,b}$ derived from her passphrase. Upon login, a session is created on the server and associated with a fresh session identifier $\mathsf{sid}_{u,b}$ sent back to the browser as a cookie. The browser sends back the cookie with every subsequent request, so the server can correlate all the user actions on the website even if these actions were taken in separate tabs, over different HTTPS connections. This login protocol can be described as follows.

Web Login and Key Derivation: $\mathsf{Login}(u,p,b)$

	user on browser a navigates to `https://b/login`
	a and b establish TLS connection c: $\mathsf{TLS}_c^{\rightarrow b}(-)$, $\mathsf{TLS}_c^{\leftarrow b}(-)$
1. $a \rightarrow b$	$\mathsf{TLS}_c^{\rightarrow b}(\mathsf{Request}(\texttt{/login}))$
2. $b \rightarrow a$	$\mathsf{TLS}_c^{\leftarrow b}(\mathsf{Response}(\mathsf{LoginForm}))$
	user enters username u and passphrase p
	a derives and stores $K = \mathsf{kdf}\ p\ A_u\ \mathsf{iter}$, $K' = \mathsf{kdf}\ p\ B_u\ \mathsf{iter}$
	a derives $\mathsf{secret}_{u,b} = \mathsf{kdf}\ p\ C_u\ \mathsf{iter}$
3. $a \rightarrow b$	$\mathsf{TLS}_c^{\rightarrow b}(\mathsf{Request}(\texttt{/login}, \texttt{user} = \texttt{u\&secret} = \mathsf{secret}_{u,b}))$
	b verifies that $\texttt{secret} = \mathsf{secret}_{u,b}$
	b generates a cookie $\mathsf{sid}_{u,b}$
	b stores $(\mathsf{sid}_{u,b}, u)$
4. $b \rightarrow a$	$\mathsf{TLS}_c^{\leftarrow b}(\mathsf{Response}[\mathsf{sid}_{u,b}](\mathsf{LoginSuccess}()))$
	a stores $(b, \mathsf{sid}_{u,b})$

We write $\mathsf{Response}[\mathsf{sid}_{u,b}](\mathsf{LoginSuccess}())$ to mean that the server sends an HTTP response with a header containing the cookie $\mathsf{sid}_{u,b}$ and a body containing the

page representing successful login. All subsequent requests from the browser to the server will have this cookie attached to it, written $\mathsf{Request}[\mathsf{sid}_{u,b}](\cdots)$.

Browser-based Cryptography and Key Storage. The login protocol above and the subsequent actions of the client role a of the encrypted storage protocol require a to generate keys, store them, and use them in cryptographic operations. To execute this logic in a browser, typical websites use JavaScript, either as a script embedded in web pages or in an isolated browser extension. In some applications, the cryptography is also implemented in JavaScript (e.g. LastPass). In others, the cryptography is provided by a Java applet but invoked through JavaScript (e.g. ConfiChair). In both cases, the keys must be stored in a location accessible to the script. Sometimes such cryptographic materials are stored in the browser's localStorage which provides a private storage area to each website and to each browser extension.

When the performance or reliability of JavaScript is considered inadequate, a few storage applications (such as SpiderOak) instead cache decryption keys on the server and perform all decryptions on the server side; these keys are discarded upon logout. In the rest of this paper, we generally assume that all cryptography is implemented on the client unless explicitly specified.

Releasing plaintext to authorized websites In addition to update and synchronize, some storage services offer advanced sharing mechanisms. For example, password managers offer a *form fill* feature whereby user data is automatically retrieved, decrypted, and released to authorized websites. This feature is implemented by a browser extension or bookmarklet and activated when a user visits a login page; the extension automatically fills the login form with the user's credentials for that page. In the protocol description below, the encrypted storage client holding the database and its decryption keys is the browser extension x.

Automatic Form Filling for Web Login: $\mathsf{Fill}(b)$

	user on browser a navigates to `https://b/login`
	a and b establish TLS connection c: $\mathsf{TLS}_c^{\rightarrow b}(-)$, $\mathsf{TLS}_c^{\leftarrow b}(-)$
1. $a \rightarrow b$	$\mathsf{TLS}_c^{\rightarrow b}(\mathsf{Request}(\texttt{/login}))$
2. $b \rightarrow a$	$\mathsf{TLS}_c^{\leftarrow b}(\mathsf{Response}(\mathsf{LoginForm}))$
	a triggers browser extension x with the current page hostname
3. $a \rightarrow x$	$\mathsf{Lookup}(b)$
	x looks up encdb for $(b,\mathsf{e},\mathsf{h})$
	x checks that $\mathsf{mac}\ K'\ (b,\mathsf{e}) = \mathsf{h}$
	x computes $(u,p) = \mathsf{decrypt}\ K\ e$
4. $x \rightarrow a$	$\mathsf{Result}(b,u,p)$
	a fills LoginForm with (u,p)

Sharing with a web link. File storage services often allow a user to share a file or folder with others, even if they do not have an account with the service. This works by sending the recipient a web link that contains within it the decryption key for the shared file. The receiver can access the file by following the link.

URL-based File Sharing: Share(u,m)

	user u sends to v the link U=`https://b/?user=`u`&file=`m`&key=`K
	user v on browser a navigates to U
1. $a \to b$	$\mathsf{TLS}_c^{\to b}$(Request[](U))
	b retrieves storage$[u] = (m, e, h)$
	b decrypts $f =$ decrypt K e
2. $b \to a$	$\mathsf{TLS}_c^{\leftarrow b}$(Response[](Download(f)))

Sending decryption keys in plaintext links is clearly a security risk since the key can easily be leaked. As a result, even services that offer link-based sharing do not use the same key for shared files as they do for private files. For instance, SpiderOak creates a fresh encryption key for each shared folder and re-encrypts its contents. When the owner needs to access and decrypt her own shared files, she must first retrieve this shared key from the server. We model this protocol in more detail in Section 4. Other applications such as Wuala or CloudFogger use a more secure sharing scheme that relies on a public key infrastructure, allowing the decryption key to be sent wrapped under the recipient's public key.

2.3 Web Attacker model

An encrypted storage application that uses JavaScript and cookie-based sessions is exposed to, and must protect against, a range of web attack vectors.

Code delivery. In typical website deployments, the JavaScript code that performs client-side encryption is itself downloaded from the web. If the attacker controls the server hosting the JavaScript, he may corrupt the application code in order to leak keys back to himself. Alternatively, if the code is downloaded over plain HTTP, a network attacker may tamper with the script.

XSS. In its simplest form, an attacker may be able to exploit unsanitized user input in the application to inject JavaScript that gets inlined in the website HTML and run along with trusted JavaScript. This may give the attacker complete control over a web page in the browser and to all cryptographic materials available to that page. Even carefully written security-conscious applications, such as Dropbox, LastPass, and ConfiChair, may still contain such weaknesses, as we show in Section 5. New browser security mechanisms are being proposed to address this issue [23].

Session Hijacking. Once a session is established, the associated cookie is the only proof of authentication for further actions. If an attacker gets hold of the session cookie, he can perform the same set of operations with the server as the user. In Section 5 we describe attacks of this kind that we found in several applications (including ConfiChair), even if they normally use HTTPS. A solution is for applications to set the cookie in *secure mode*, disallowing the browser to send it over an unencrypted connection.

CSRF. When an action can be triggered by accessing some URL, for example changing the current user's email address or his role in the session, a malicious

site can force its users to access this URL and perform the action on their behalf, with attacker-controlled parameters. Although it is up to the application to prevent these kind of attacks, various varieties of CSRF remain common, even in security-oriented web services [9]. A common solution is to use an unguessable authorization token bound to the user session and require it to be sent with every security-sensitive request.

Phishing and Open Redirectors. Features involving third parties may introduce new attack vectors. For instance, in the automatic form filling protocol above, an untrusted website may try feeding the extension a fake URL instead of the legitimate login URL, to trick the extension into retrieving the user's password for a different website. Similarly, open redirectors such as URL `http://b/?redir=x`, that redirect the user to an external website `x`, facilitate phishing attacks where the website `x` may fool users into thinking that they are visiting a page on `b` when in fact they are on website `x`.

In summary, the design of cryptographic web applications must account for prevalent web vulnerabilities, not just the formal cryptographic attacker of Section 2.1. Next, we introduce our methodology for analyzing such applications.

3 Automated Verification of Web Cryptography

We describe the WebSpi library for ProVerif, and discuss how it is used to model and verify web applications. We show our extensions to WebSpi to model new JavaScript-based attacks. For details on ProVerif, see the official manual [14].

3.1 Processes

The language underlying ProVerif is a variant of applied pi-calculus [3]. Computations are described as the interaction of message-passing processes that communicate over asynchronous named channels. Knowing the name of a channel is enough to be able to send or receive messages on it. The name of a channel defined as private to a process cannot be guessed by other processes, so the creator controls its scope (that can be extended by sending the channel name to other processes). Processes have access to local databases where they can store and retrieve messages. Atomic messages, typically ranged over by $a, b, c, h, k, ...$ are tokens of basic types. Basic types are channels, bitstrings or user-defined. Messages can be composed by pairing (M, N) or by applying n-ary data constructors and destructors $f(M_1, ..., M_n)$. Constructors and destructors are particularly useful for cryptography, as described below. Pattern matching $= M$ is extensively used to parse messages.

ProVerif models *symbolic* cryptography: cryptographic algorithms are treated as perfect black-boxes whose properties are abstractly encoded using constructors and destructors. Consider authenticated encryption:

```
fun aenc(bitstring,symkey): bitstring.
reduc forall b:bitstring,k:symkey; adec(aenc(b,k),k) = b.
```

Given a bit-string b and a symmetric key k, the term aenc(b,k) stands for the bitstring obtained by encrypting b under k. The destructor adec, given an authenticated encryption and the original symmetric key, evaluates to the original bit-string b. ProVerif constructors are collision-free (one-one) functions and are, by default, only reversible if equipped with a corresponding destructor. Hence, MACs and hashes are modeled as irreversible constructors, and asymmetric cryptography is modeled using public and private keys:

```
fun hash(bitstring) : bitstring.
fun pk(privkey):pubkey.
fun sign(bitstring,privkey): bitstring.
reduc forall b:bitstring,sk:privkey; verify(sign(b,sk),pk(sk)) = b.
```

These and other standard cryptographic operations are part of the ProVerif library. Users can define other primitives where necessary. Such primitives can be used for example to build detailed models of applications like ConfiChair [7].

The WebSpi library defines data types related to the HTTP protocol and provides interfaces to the core functionality of browsers and web servers, in the form of a set of private channels. Application-layer protocols are expressed as processes linked to this channel interface. The rest of the network, including potential attackers, can be thought of as arbitrary processes with access to net and any other public channel.

3.2 WebSpi Architecture

In our model, *users* surf the web by interacting with web *pages* on *browsers* that communicate on the public channel net over HTTP(S) with *servers* that host web *applications*.

Users. Users are endowed with, or can acquire, username/password credentials to access applications. Applications are identified by a host name and a path within that host. The behaviour of specific web page users can be modeled by defining a UserAgent process that uses the browser interface described below.

Servers. Servers possess private and public keys used to implement encrypted TLS connections with browsers. These are stored in the serverIdentities table together with the server name (protocol and host) and a flag xdr specifying if cross-domain requests are accepted. The WebSpi implementation of a server is given by the HttpServer process below. HttpServer handles HTTP(S) messages (and encryption/decryption when necessary) and routes parsed messages to the corresponding web applications on the channels httpServerRequest and httpServerResponse. To model the server-side handler of a web application one needs to write a process that uses this interface to send and receive messages.

```
let HttpServer() =
  in(net,(b:Browser,o:Origin,m:bitstring));
  get serverIdentities(=o,pk_P,sk_P,xdr) in
  let (k:symkey,httpReq(u,hs,req)) = reqdec(o,m,sk_P) in
  if origin(u) = o then
```

```
let corr = mkCorrelator(k) in
out(httpServerRequest,(u,hs,req,corr));
in(httpServerResponse,(=u,resp:HttpResponse,cookieOut:CookiePair,=corr));
out(net,(o,b,respenc(o,httpResp(resp,cookieOut,xdr),k))).
```

Browsers. Each browser has an identifier b and is associated with a user. The WebSpi implementation of a browser is given by the HttpClient process (we inline some fragments below). Cookies and local storage are maintained in global tables indexed by browser, page origin and, only for cookies, path. JavaScript running on a page can access cookies and storage associated with the page origin using the private channels getCookieStorage and setCookieStorage, in accordance to the Same Origin Policy. Cookies can be flagged as *secure* or *HTTP-only*. Secure cookies are sent only on HTTPS connections and HTTP-only cookies are not exposed to pages via the CookieStorage channel. For example, the HttpClient code that gets triggered when the JavaScript of page p on browser b wants to set cookies dc and store ns in local storage is:

```
in (setCookieStorage(b),(p:Page,dc:Cookie,ns:Data));
get pageOrigin(=p,o,h,ref) in get cookies(=b,=o,=h,ck) in
insert cookies(b,o,h,updatedomcookie(ck,securejs(dc),insecurejs(dc)));
insert storage(b,o,ns)
```

Here, the function updatedomcookie prevents JavaScript from updating the HTTP-only cookies of the cookie record ck.

The main role of the browser process is to handle requests generated by users and web pages, and their responses. The location bar is modeled by channel browserRequest, which can be used by to navigate to a specific webpage. Location bar request have an empty referrer header. Hyperlink clicks or JavaScript GET/POST requests are modeled by the pageClick channel. The browser attaches relevant headers (referrer and cookies) and sends the request on the network. When it receives the response, it updates the cookies and creates a new page with the response data. Process HttpClient also takes care of encrypting HTTPS requests, decrypting HTTPS responses, and handling redirection responses. AJAX requests are sent to the browser on channel ajaxRequest. When the browser receives the response to an AJAX request it passes on the relevant data to the appropriate web page. (Although we abstract away the tree-like structure of the DOM, we do represent its main features salient to modeling web interactions: cookies, hyperlinks, location bar, forms, etc.) We give the HttpClient code for sending a request req to URI u from page p, with referrer ref and AJAX flag aj:

```
let o = origin(u) in let p = path(u) in
get cookies(=b,=o,=slash(),cs) in get cookies(=b,=o,=p,cp) in
let header = headers(ref, cookiePair(cs,cp), aj) in
get publicKey(=o,pk_host) in
let m = httpReq(u,header,req) in
let (k:symkey,e:bitstring) = reqenc(o,m,pk_host) in
out(net,(b, o, e));
```

The request header is obtained concatenating the referrer, the cookies cs for path "/" and cp for path p and the AJAX flag aj. If needed one could extend

the model by including additional headers such as `Origin` [9]. Note how the code retrieves the public key pk_host of the destination server, which is used to create the symmetric key k and the encrypted message e. The origin parameter o passed to the encryption function reqenc specifies if the chosen protocol is HTTP or HTTPS. In the former case, e equals m.

To model the client side of a web application, one needs to write a process that can access the private browser interface channels pageClick, ajaxRequest, getCookieStorage and setCookieStorage.

Web Attacker Model. Representing the network as a public channel net enables the standard Dolev-Yao *network attacker*, that can intercept and inject messages but is not able to break cryptography. To model a *compromised server*, we simply release its private key on a public channel so that an arbitrary attacker process can impersonate the server. We enable XSS and *code injection* attacks by defining a process AttackerProxy that receives messages on a public channel (available to the attacker) and forwards them on the browser's private channels. The parameters sent on these channels include the browser and page ids, which are normally secret. We can selectively enable the compromise of a specific page on a specific browser by releasing the corresponding ids to the environment. CSRF attacks are enabled by the willingness of the user to visit attacker websites and by the ability of our model to represent GET/POST requests and attach the corresponding cookies.

Verification in WebSpi. The verification model of WebSpi is the same as in ProVerif. Security goals in ProVerif are typically written as correspondence assertions between events embedded in the code [12]. The command event e(M1,...,Mn) inserts an *event* e(M1,...,Mn) in the trace of the process being executed. A script in fact contains processes and *queries* of the form $\forall M_1, ...M_k. e(M_1, ...M_k) \Rightarrow \phi$. ProVerif tries to prove that whenever the event e is reachable, the formula ϕ is true (ϕ can contain conjunctions or disjunctions). In Section 4 we will show concrete security queries.

The soundness properties of ProVerif [12] also hold for our security policies. If an expect is satisfied, then it is satisfied in all traces of running the applied-pi processes defined in the script in parallel with any arbitrary attacker processes. If ProVerif proves that an expect is not satisfied, it outputs a proof derivation that explains how an attacker can trigger an event that violates the policy.

Although very expressive, WebSpi is not a complete model of the web. For example, our model of the Same Origin Policy does not include `<iframe>` tags from different origins within the same page, and we do not model several HTTP headers such as `Origin` and `ETag`. Hence, our main focus is on discovering attacks, which can be validated in the real world, rather than on providing positive guarantees, which may be violated in practice due to omissions in our model.

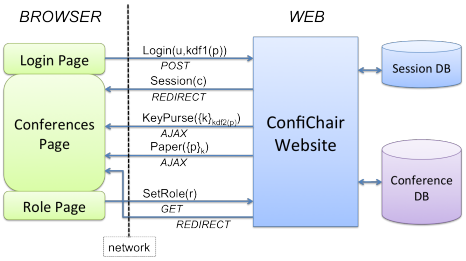
4 Analyzing Encrypted Web Storage Services

In this section, we analyze three web applications that use the cloud to store encrypted secrets. We show how to model these applications using WebSpi and

verify them using ProVerif against realistic web attackers. We show how web vulnerabilities enable concrete attacks that leak secrets to a web attacker. It is difficult to completely eradicate such vulnerabilities from complex, real-world web applications. For that reason we propose countermeasures that harden such applications even in the presence of vulnerabilities.

4.1 ConfiChair

ConfiChair [7] is a cloud-based conference management system that seeks to offer stronger security and privacy guarantees than current systems like EasyChair and EDAS. Each conference has a chair, authors, and a program committee (of reviewers). Once a user logs in at the login page, she is forwarded to a Conferences page where she may choose a conference to participate in. The user may choose her role in the conference by clicking on "change role" which forwards her to the role page. Papers and reviews are stored encrypted on the web server, and each user holds keys to all papers and reviews she is allowed to read in a *keypurse*. For example, each paper has an encryption key (generated by the author) that is stored in the author's and conference chair's keypurses. Each conference has a private key stored only in the chair's keypurse and a shared reviewer key that is stored in each reviewer's keypurse. Each user's keypurse is also stored encrypted on the web server under a key derived from her password. The password itself is not stored there, instead a separate key derived from the password is used to authenticate the user. The web server authenticates users before sending them their keypurses and enforces role-based access control to conference actions and per-user access control to papers and reviews. All the cryptography for decrypting and encrypting keypurses, papers, and reviews is performed in the browser using a combination of JavaScript and a Java applet.



WebSpi Analysis. We model and evaluate paper downloads using WebSpi.

Login. We model the login page using two processes: LoginApp represents a server-side webpage listening for requests on `https://confichair.org/login`, and LoginUserAgent represents the client-side JavaScript and HTML downloaded from this URL. These processes implement the web login protocol of Section 2.2, but do not yet derive the encryption and MAC keys.

The process LoginUserAgent downloads a login form, waits for the user to type her username and password, derives an authentication credential from the password and sends the username and credential to LoginApp over HTTPS (through the network channel between the browser and HTTP server processes):

```
let loginURI = uri(https(), confichair, loginPath(), nullParams()) in
out(browserRequest(b),(loginURI, httpGet()));
in (newPage(b),(p:Page,=loginURI,d:bitstring));
```

```
get userData(=confichair, uid, pwd, paper) in
let cred = kdf1(pwd) in
in (getCookieStorage(b),(=p,cookiePair(cs,ch),od:Data));
out (setCookieStorage(b),(p,ch,storePassword(pwd)));
event LoginInit(confichair, b, uid);
out(pageClick(b),(p,loginURI,httpPost(loginFormReply(uid,cred))))
```

Notably, the process stores the password in the HTML5 local storage corresponding to the current origin `https://confichair.org`, making it available to any page subsequently loaded from this origin. When the user logs out, the local storage is purged.

The server process LoginApp is dual to the LoginUserAgent. It checks that the credential provided by the user in the login form is valid (by consulting a server-side database modeled as a table) and creates a session id passed to the browser as a cookie for all pages on the website, before redirecting the user to the conferences page.

Paper Download. We model all the conference pages using a server-side process ConferenceApp and a client-side process ConferenceUserAgent. The process ConferencesUserAgent first makes an AJAX request to retrieve the encrypted keypurse of the logged in user. It then decrypts the keypurse using a key derived from the cached password and stores the decrypted keypurse in local storage for the current origin (`https://confichair.org`).

```
let keypurseURI = uri(https(), confichair, keyPursePath(), nullParams()) in
out (ajaxRequest(b),(p,keypurseURI,httpGet()));
in (ajaxResponse(b),(=p,=keypurseURI,JSON(x)));
in (getCookieStorage(b),(=p,cookiePair(cs,ch),storePassword(pwd)));
let keypurse(k) = adec(x, kdf2(pwd)) in
out (setCookieStorage(b),(p,ch,storeKeypurse(k)))
```

For simplicity, the keypurse contains a single key, meant for decrypting the current user's papers. Subsequently, the user may at any point ask to download a paper and decrypt the downloaded PDF with the keypurse.

```
let paperURI = uri(https(), h, paperPath(), nullParams()) in
out (ajaxRequest(b),(p,paperURI,httpGet()));
in (ajaxResponse(b),(=p,=paperURI,JSON(y)));
in (getCookieStorage(b),(=p,cookiePair(cs,ch),storeKeypurse(k)));
let paper = adec(y,k) in event PaperReceived(paper))
```

Security Goals. We model two simple security goals for our ConfiChair website model. First, the login mechanism should authenticate the user. This is modeled as a correspondence query:

```
event(LoginAuthorized(confichair,id,u,c)) ⟹event(LoginInit(confichair,b,id))
```

Second, that a user's papers must remain syntactically secret. We model this using an oracle process that raises an event when the attacker successfully guesses the contents of a paper

```
in(paperChannel, paper:bitstring);
get userData(h, uId, k, =paper) in event PaperLeak(uId,paper).
```

We then ask whether the event PaperLeak is ever reachable. The queries written here are quite simple. More generally, they must account for compromised users whose passwords are known to the attacker. For the login and conferences processes above, these queries do indeed hold against an adversary who controls the network, some other websites that honest users may visit, and some set of compromised users.

Attacker Model: XSS on Role Page. Our security analysis found a number of web vulnerabilities. Here we describe how the change-role functionality on the ConfiChair webpage is vulnerable to an XSS attack. If an attacker can trick a user into visiting the URL `http://confichair.org/?set-role=<script>S</script`, ConfiChair returns an error page that embeds the HTML tag `<script>S</script>`, causing the tainted script `S` to run. We model this attack as part of the client-side process RoleUserAgent for the role page: after loading the page, the process leaks control of the page to the adversary by publicly disclosing its identifier:

```
let roleURI = uri(https(), h, changeRolePath(), roleParams(x)) in
out(browserRequest(b),(roleURI, httpGet()));
in (newPage(b),(p:Page,=roleURI,y:bitstring));
out(pub, p)
```

The attacker may subsequently use this page identifier p to make requests on behalf of the page, read the cookies, and most importantly, the local storage for the page's origin.

Attacks on Authentication and Paper Secrecy. If we add this RoleUserAgent to our ConfiChair model ProVerif finds several attacks against our security goals. First, the XSS attacker may now read the current user's password from local storage and send it to a malicious website. This breaks our authentication goal since from this point onwards the attacker can pretend to be the user. Second, the XSS attacker may read the current user's keypurse from local storage and send it to a malicious website. This breaks our paper secrecy goal since the attacker can decrypt the user's papers.

These attacks have been experimentally confirmed on the ConfiChair website (along with some others described in Section 5). They break the stated security goals of ConfiChair by leaking the user's papers and reviews to an arbitrary website. The previous ProVerif analysis of ConfiChair [7] did not cover browser-based key management or XSS attacks: its security proofs remain valid in the cloud-based attacker model.

Mitigations and Countermeasures. An obvious mitigation is to eliminate the XSS attack on the change-role functionality. A more interesting design question is how to change the ConfiChair website to be more robust in the presence of such XSS attacks. We focus on countermeasures that keep the current workflow.

First, there is no need for the website to store the cleartext password in local storage, where an XSS attacker can obtain it. Storing just the decryption key is enough. Second, we propose to use a fresh session-specific wrapping key to encrypt both the decryption key and the keypurse before storing them in local storage. The website can then decide which pages need access to these keys and

expose the wrapping key in a secure cookie only for those pages. For example, suppose all pages that need access to the wrapping key are served from the sub-domain `secure.confichair.org`, whereas all other pages are served from the parent domain `confichair.org`. The wrapping key can then be set as a cookie for the sub-domain, pages in the parent domain will not be able to access it. In this design, the website never has both the key and the encrypted data. During login the browser has the password and the website has the encrypted data. After login, the browser has a re-encrypted keypurse and the website has the fresh encryption key. With these changes our secrecy and authentication queries are verified by ProVerif. That is, if the login and conferences pages are hosted on the secure sub-domain and are XSS-free, then XSS attacks on other pages do not impact the security of the application. Whether this countermeasure is practical or even resistant to more sophisticated iframe-based attacks requires further investigation.

4.2 SpiderOak

SpiderOak is a commercial cloud-based backup, synchronization and sharing service. It advertises itself as "zero-knowledge", that is, the SpiderOak servers only store encrypted data, but never the associated decryption keys. Users typically use downloaded client software to connect to SpiderOak and synchronize their local folders with cloud-based encrypted backups. However, SpiderOak also provides its users with a web front end to access their data so that they can read or download their files on a machine where they have not installed SpiderOak.

When a user logs into the SpiderOak website, her decryption keys are made available to the web server so that it can decrypt a user's files on her behalf. These keys are to be thrown away when the user logs out. However, if the user shares a folder using a web link with someone else, the decryption key is treated differently. The key is embedded in the web link, and it is also stored on the website for the file owner's use. We focus on modeling this management of shared folders (called shared rooms) on SpiderOak.

WebSpi Analysis. The SpiderOak login process is similar to ConfiChair, except that besides the derived authentication credential it sends also the plaintext password to the server. After login, the user is forwarded to his root directory, from where he may choose to open one of his shared folders (called shared rooms).

The process SharedRoomUserAgent models the client-side JavaScript triggered when the user accesses a shared folder. It makes an AJAX request to retrieve the URL, file names, and decryption key for the folder. It then constructs a web link consisting of the URL, file name, and the decryption key and uses the URL-based sharing protocol of Section 2.2 to retrieve its files. The server-side process SharedRoomApp responds to the AJAX request from the user: it authenticates the user based on her login cookie, retrieves the folder URL, file names, and decryption key from a database and sends it back in a JSON formatted message. It also responds to GET requests for files, but in this case the user does not have to be logged in; she can instead provide the name of the file and the decryption key as parameters in the URI.

Similarly to ConfiChair, we set two security goals: user authentication and syntactic file secrecy. ProVerif is able to show that our SpiderOak model preserves login authentication but it fails to prove file secrecy as we explain below.

JSONP CSRF Attack on Shared Rooms. The SpiderOak shared rooms page is vulnerable to a CSRF attack on its AJAX call for retrieving shared room keys. If a user visits a malicious website while logged into SpiderOak, that website can trigger a cross-site request to retrieve the shared room key for the currently logged-in user. The browser automatically adds the user's login cookie to the request and since the server relies only on the cookie for authentication, it will send back the JSON response to the attacker. The attacker can then retrieve the file by constructing a web link and making a GET request.

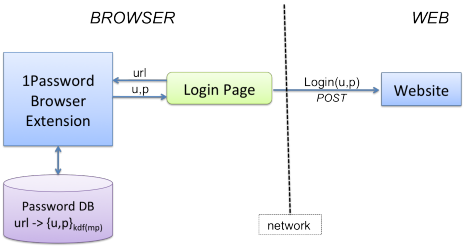
This CSRF attack only works if the target website explicitly enables cross-domain AJAX requests, as we found to be the case for SpiderOak. In our SpiderOak model, the SharedRoomsApp page sets the xdr flag, and ProVerif finds the CSRF attack (as a violation of file secrecy).

Mitigations and Countermeasures. We experimentally confirmed the attack on the SpiderOak website and on our advice, SpiderOak removed cross-domain access to shared rooms. As in ConfiChair, we consider whether a different design of SpiderOak would make it resistant to attack even if it had a CSRF vulnerability.

One countermeasure is to encrypt the shared room key with the owner's password. Hence, only the owner can decrypt the key, but that is adequate since other shares are given the key in the web link anyway. ProVerif shows that with this fix the attacker is no longer able to obtain the file, even though the CSRF attack is still enabled. The attacker can get the file URL but not the key.

4.3 1Password

1Password is a password manager that uses the cloud only as an encrypted store. Typically, it uses Dropbox to backup and replicate a user's encrypted password database. To protect these passwords in transit, on Dropbox, and on each device, the password database is always encrypted on the client before uploading. Even though 1Password does not host any website, we show that it is nonetheless vulnerable to web-based attacks.



Password managers such as 1Password provide a browser extension that makes it easier for users to manage their passwords. The first time a user visits a login page and enters his password, the browser extension offers to remember the password. On future visits, 1Password offers to automatically fill in the password. Concretely, the extension looks at the origin of the page and uses it to lookup its database. If a password is found, it is decrypted and filled into the login form.

WebSpi Analysis. We model 1Password and its browser extension as a process that waits for messages from a page on a channel extensionChannel; it then looks

Name	Alternate Login	Insecure Cookie	XSS	CSRF	Open Redirector	Frameable
Dropbox	OAuth	✗	✓	✓	✗	✗
SpiderOak	HTTP Auth	✗	✗	✓	✗	✗
LastPass	YubiKey	✗	✓	✓	✗	✗
PassPack	YubiKey	✗	✗	✓	✗	✓
ConfiChair	None	✓	✓	✓	✓	✓
Helios	OAuth, OpenID	✓	✓	✓	✓	✓

Table 2. Web vulnerabilities in cloud storage websites

for an entry for the current origin in the password database (called a keychain store). If it finds an entry, it asks the user for a master password, uses it to decrypt the username and password, and returns them on the extension channel to the requesting page. This protocol corresponds to the automatic form filling protocol of Section 2.2, except that 1Password does not include a MAC with the encrypted data. We compose this extension process with a standard login application, for example, as in the SpiderOak model, to obtain a simple model for 1Password. Login authentication and password secrecy are the security goals.

Metadata Tampering on the Password Database. 1Password is designed to be resistant to attacks on Dropbox and to an attacker who has stolen a user's device. We model an attacker with read/write access to the encrypted password database. Each password entry in 1Password is stored as a separate text file in Dropbox, so our model captures attackers who can read or write to these files. When composed with this attacker and a malicious website, ProVerif finds that password secrecy is violated (hence, so is login authentication).

The attack proceeds as follows: the attacker reads the entry for (say) SpiderOak from the database and replaces the hostname SpiderOak with the name of his own server, Mallory. Since the origin is not encrypted or integrity-protected in the database, this modification remains undetected. The next time the user visits Mallory's website, the page requests a password for Mallory and the 1Password extension instead provides the password for SpiderOak, which gets leaked to Mallory. We call this attack a metadata tampering attack since the attacker manages to modify the metadata surrounding an encrypted password. Similar attacks are applicable in other storage services.

Mitigations and Countermeasures. The metadata tampering attack only applies if the attacker has write access to the encrypted database. Hence, one countermeasure is to make the database inaccessible to the attacker. A more robust solution is to add metadata integrity protection to the password database. As in the protocols of Section 2.2, we propose that both the ciphertext and all metadata in a keychain should be MACed with a key derived from the master password. ProVerif verified that this prevents metadata tampering, and hence password leaks, even if the password database is stored in an insecure location.

5 Concrete Attacks on Encrypted Web Storage Services

We have shown how to formally analyze core components of three encrypted web storage services using WebSpi and ProVerif. In each case, we found that

the security provided by cryptography was circumvented by a web-based attack. For illustration, Table 2 summarizes vulnerabilities on storage websites found by us and by others. Besides XSS and CSRF, this table notes websites that did not use secure cookies and were thus vulnerable to session hijacking, those that had open redirectors that may lead to phishing, and those that were framable and thus vulnerable to clickjacking. These vulnerabilities are ubiquitous on the web and seem difficult to avoid on realistic websites. We now explain the impact of such vulnerabilities on our target applications. All the attacks below were discovered and reported by us, either during this work, or in [11].

Metadata Tampering. Encrypted storage services such as BoxCryptor, Cloudfogger, and 1Password aim to be resilient to the tampering of encrypted data on DropBox. However, these applications failed to protect metadata integrity, so an attacker could confuse users about their stored data. For example, one could rename an encrypted file in BoxCryptor and replace an encrypted file in CloudFogger without these modifications being detected.

User Impersonation. Both ConfiChair and Helios can be attacked if a logged-in user visits a malicious website. If a logged-in conference chair visits a malicious website, the website may use a series of CSRF and clickjacking attacks to close submissions or release referee reports to authors. On Helios, the problem is more serious. If a user authenticates on Helios using Facebook (a common usage pattern), any malicious website she subsequently visits may steal her authentication token and impersonate her, even if she logged out of Helios. The attack relies on an open redirector on Helios and the OAuth 2.0 protocol implemented by Facebook, and corresponds to a token redirection attack previously found using WebSpi [8]. This attack undermines voter authentication on Helios, and lets an attacker modify election settings by impersonating the election administrator.

Password Phishing. Password managers are vulnerable to a variety of phishing attacks where malicious websites try to fool them into releasing passwords for trusted websites. Metadata tampering, as shown for 1Password, also applies to Roboform. Another attack vector is to use carefully crafted URLs that are incorrectly parsed by the password manager. A typical example is `http://a:b@c:d`, which means that the user a with password b wants to access website c at port d, but may be incorrectly parsed by a password manager as a user accessing website a at port b. We found such vulnerabilities in 1Password and many popular JavaScript URL parsing libraries. We also found that password managers like LastPass that use bookmarklets are vulnerable to JavaScript rootkits [5].

6 Conclusions

In this paper, we formally analyzed 3 encrypted web storage applications, and described concrete security attacks in 7 more. Our reports resulted in security updates for Wuala, 1Password, LastPass, and SpiderOak, and security advisories for the ConfiChair and Helios websites, others are being discussed. WebSpi is a useful tool for evaluating web applications and for experimenting with their

design to make them more resilient to standard web vulnerabilities. As WebSpi is not complete, we leave the task of modeling even more attacks, such as framing [22], JavaScript rootkits [5], and other scenarios [1], to future work.

References

1. Browser security handbook. `http://code.google.com/p/browsersec`.
2. How secure is Dropbox? `https://www.dropbox.com/help/27/en`.
3. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. *SIGPLAN Not.*, 36:104–115, January 2001.
4. B. Adida. Helios: Web-based open-audit voting. In *USENIX Security Symposium*, pages 335–348, 2008.
5. B. Adida, A. Barth, and C. Jackson. Rootkits for JavaScript environments. In *Workshop on Offensive Technologies (WOOT)*, 2009.
6. D. Akhawe, A. Barth, P.E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *CSF*, pages 290–304, 2010.
7. M. Arapinis, S. Bursuc, and M. Ryan. Privacy supporting cloud computing: Confichair, a case study. In *POST*, pages 89–108, 2012.
8. C. Bansal, K. Bhargavan, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *CSF*, pages 247–262, 2012.
9. A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *CCS*, pages 75–88, 2008.
10. A. Belenko and D. Sklyarov. "Secure Password Managers" and "Military-Grade Encryption" on Smartphones: Oh, Really? Technical report, Elcomsoft Ltd., 2012.
11. K. Bhargavan and A. Delignat-Lavaud. Web-based attacks on host-proof encrypted storage. In *Workshop on Offensive Technologies (WOOT)*, 2012.
12. B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.
13. B. Blanchet and A. Chaudhuri. Automated formal analysis of a protocol for secure file sharing on untrusted storage. In *IEEE Symposium on Security & Privacy*, 2008.
14. B. Blanchet and B. Smyth. *ProVerif: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*. `http://www.proverif.inria.fr/manual.pdf`.
15. Aaron Bohannon and Benjamin C. Pierce. Featherweight Firefox: Formalizing the core of a web browser. In *WebApps*, 2010.
16. T. Groß, B. Pfitzmann, and A. Sadeghi. Browser model for security analysis of browser-based protocols. In *ESORICS*, pages 489–508, 2005.
17. E. Hammer-Lahav, D. Recordon, and D. Hardt. The OAuth 2.0 Authorization Protocol. IETF Internet Draft, 2011.
18. D. Jackson. Alloy: A logical modelling language. In *ZB*, page 1, 2003.
19. Seny Kamara and Kristin Lauter. Cryptographic cloud storage. In *Financial cryptograpy and data security*, pages 136–149, 2010.
20. J. Kelsey, B. Schneier, C. Hall, and D. Wagner. Secure applications of low-entropy keys. In *ISW '97*, pages 121–134, 1998.
21. E. Rescorla. HTTP over TLS, 2000. Request for Comments 2818, IETF.
22. G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *Web 2.0 S&P*, 2010.
23. B. Stearne and A.Barth eds. Content Security Policy 1.0. W3C Working Draft, 2012.
24. S. Yoshihama, T. Tateishi, N. Tabuchi, and T. Matsumoto. Information-Flow-Based Access Control for Web Browsers. *IEICE Transactions*, E92-D(5):836–850, 2009.